Pidgin—A Systems Programming Language

DDJ is pleased to present the first of two articles by Dr. William Gale (the second article will appear next month). In these articles Dr. Gale describes three unusual pieces of software: Tincmp, a clever macro processor; Pidgin, a systems programming language; and Meta4, a compiler-compiler.

Dr. Gale calls Pidgin a "low-level language." This is only justified by its small number of data types (bytes, words, and vectors of bytes or words). Pidgin's assortment of control statements is as good as that of many high-level languages. Despite this, Pidgin is simple enough that it can be fully implemented as a set of Tincmp macros.

Tincmp, the macro processor, is the key to the series. Dr. Gale illustrates its power by using it as a translator from Pidgin source code to 6502 machine language. We're sure that isn't the end of its uses. Tincmp is given here as a program written in Pidgin. The bootstrap procedure is to paraphrase that listing in BASIC or Pascal, feed the Pidgin macros and the Tincmp source program to that program, and receive a machine-language version of Tincmp as output.

Next month DDJ will publish Dr. Gale's description of Meta4, a compiler-compiler (a program that takes a formal language definition as input and delivers a compiler for that language as output). Meta4 is a development of Meta II, described in DDJ #44. The source for Meta4 will be given in Pidgin. Those readers who've brought up Tincmp by then will be able to implement Meta4 at once.

We hope DDJ's readers will be as excited by Tincmp, Pidgin, and Meta4 as we are. DDJ welcomes contributions based on Dr. Gale's articles. Some possibilities: a listing of Tincmp in BASIC or Pascal, or Tincmp macros to translate Pidgin to another machine language or to another programming language, or reports of your application of any of the three programs to other problems.

Pidgin is a low-level, machine-independent programming language. Because it is easily compiled, it is suitable for systems programming applications where portability is the foremost requirement. The original design purpose was cross-time portability, so work done on my current machine could be cumulative for my next machine. But it should also be suitable for describing to a Z80 what I've done on my 6502. People with a homebrew system that nobody else writes software for may be particularly interested in a pool of machine-independent software.

Pidgin can be compiled by a macro processor that recognizes nine paramenters each of length one. Such a macro processor is easy to write, the more difficult part being to write the macros for the 80 statements of Pidgin. A macro processor written in Pidgin is included with this article. It takes 500 lines with 8000 characters including comments, and compiles into 3.5 kilobytes (with another 5 kilobytes for macro storage).

Next month another Pidgin program will be described, the Meta4 compiler generator. A compiler generator is a program that makes it easier to write a compiler. Meta4 will thus allow defining more powerful languages than Pidgin, with a machine-independent compiler. It is my hope in presenting Pidgin and these programs that some of you Z80 and 6800 users will write and publish Pidgin programs that I can use on my 6502. Microcomputers are software limited, and the bigger the pool of users and writers, the bigger our pool of software will be.

by William A. Gale

Brief Description of Pidgin

The 80 statements of Pidgin can be seen from the templates in either the first or second macro sets included. The discussion of Tincmp, later in this article, will clarify how these macros are to be read. The templates start with six declaration statements. The declarations set aside storage locations for each of the two data types and the four combinations of vector types. For the vectors, the declarations give the length of storage to reserve. Indexing is from zero, so the maximum index is one less than the reserved spaces. BYTE variables allow indexing BYTE vectors from 0 to 255, or INT vectors from 0 to 127. INT variables allow indexing at least from 0 to 32000.

The macros continue with fifteen data transfer statements. The transfer statements give the means for storing and retrieving in each of the four vector types. They also give the means for translating from BYTE to INT and vice versa. The BYTE translation of an INT is the low order bits of the INT. The transfer statements also include statements for setting BYTE and INT variables to a constant number, and for setting a BYTE variable to a character constant. (The statement XX= '2' will set XX to the constant 50 if ASCII is used internally.) Two other transfers got added late, and are at the end of the macros. PACK and UNPACK are abbreviations that are rather ugly, but I finally decided I didn't want to be without them when they are so easy to do for a micro. UNPACK moves the lowest byte in the named INT to the third argument, and the second lowest byte to the second argument. PACK copies the third argument into the lowest byte of the named INT and the second argument into the next lowest byte. If the INT is larger than 2 bytes, the remainder is zero filled.

Following the bulk of the transfer statements are the eleven arithmetic statements. The arithmetic statements include the usual four binary operations for integer variables, and addition and subtraction for BYTE variables. They also include increment and decrement operations for both types. Addition and subtraction are modulo some constant, so that (0−1) + 1=0.

There follow eleven statements to generate and manipulate logical values for byte variables. The logical statements include eight comparisons and three Boolean algebra statements. The eight comparisons each set a BYTE variable after comparing two variables of the same type for equality (==), inequality (!=), strictly less (<!), and less or equal (<=). The BYTE variable is set to zero if the relation is false. If it is true, the byte will be set with at least one particular bit non-zero. (This allows the boolean operations to be done on bytes defined by comparisons.) The three Boolean operations form the Boolean and (&), or (?), and not (!) for BYTE variables. For INT variables 0−1 <0, while for BYTE variables 0 < 0 − 1. That is, under comparison, INT acts as if it holds positive and negative numbers for some range, while BYTE acts as if it holds only positive numbers.

The following 25 statements are control statements of various types. I will discuss them in groups that are not in the same order as the macros.

The assembler control statements will be highly machine dependent, and should occur first in the program. Then they can be changed easily for a different machine. They allow definition of where to assemble the program (LOMEM), and where to keep the variables (REGISTER and HIMEM). If you need something more here, don't hesitate to add it. TOP can also be used to set up assembler conditions.

The program control statements are the most varied. The main program is marked off by BEGINMAIN and ENDMAIN. Subroutines are marked by SUB $$ and ENDSUB. On encountering a GOSUB $$ statement, control passes to the statement following the corresponding SUB $$ statement. Here the two characters can be any two alphanumeric symbols. On finding a RETURN or ENDSUB, control passes to the statement following the calling GOSUB $$ statement. When a GOTO $$ is encountered, control passes to the corresponding LOC $$ statement. Only decimal digits are legal for the parameters in LOC and GOTO pairs. This restriction to 100 labels is not a problem because of the remaining control statements.

The IF $$ statement specifies a BYTE variable with the two parameter characters. If the named BYTE variable is nonzero, control continues with the next following statement. Otherwise control passes to the statement beyond the next unmatched ELSE or ENDIF. Unmatched means that, for instance, between an IF and ELSE, you can put other complete sets of IF– (ELSE) –ENDIF statements.

The ON $$ statement also specifies a BYTE variable. If the named variable is nonzero, execution continues with the following statement. Otherwise, control passes to the statement beyond the next unmatched ENDWHILE statement. When an ENDWHILE is encountered, control passes to the first previous unmatched WHILE. So WHILE-ON-ENDWHILE is the form for all loops in Pidgin.

The CHOOSE ON sequence is a choice among variables, not among constants. The CHOOSE ON $$ statement specifies a BYTE variable, as does the CASE $$ statement. When a CHOOSE ON statement is encountered, control passes to the next CASE statement for which the two variables are equal, or else to the DEFAULT if there is one, or the statement following the ENDCHOOSE if there is no DEFAULT. After matching a CASE variable, when a CASE, DEFAULT, or ENDCHOOSE statement is encountered, control passes to the statement following the next ENDCHOOSE. (In the above specification, words specifying the nesting of CHOOSE's are omitted. I think that makes the explanation clearer. But the CHOOSE, like IF and WHILE can be nested.)

STOP $ terminates execution of the program, and makes the digit specified available to any supervisory program. A zero is interpreted as a normal end, any other digit is an error condition.

Two specific control statements are designed to let Pidgin programs speak to another one. BEGINMAIN specifies a BYTE variable AC and a BYTE indexed INT vector, IAV. AC (Argument Count) can be used to give the number of entries to be found in IAV. The key point is that these are special locations, known to all Pidgin programs. They are particularly useful for passing file references, or program addresses between programs. CALL I$$ is a kluge that will be recognized by BASIC users. It will make a subroutine call to the absolute location stored in the INT variable named.

The final group of statements has the input-output statements. Pidgin is designed so that input and output is usually character by character. WRITE $$ will output the named byte to the terminal. READ $$ will get one character from the terminal and set the named BYTE variable equal to it. The common exception to single-character input and output is MS, which will output the nine specified characters to the terminal. The single quote character (') should not occur among the parameters of MS. The remaining input-output statements deal with files.

The first statement to use is OPEN $$ FOR $$ AT I$$. The first two parameters specify an INT indexed BYTE vector to be used for a buffer. The third and fourth parameters specify a BYTE variable that contains either 'R', for read, or 'W' for write. The fifth and sixth parameters name an INT variable that gives a "block number". The specification of a buffer, and allocation of space means that any number of files can be open, so long as one buffer per file is allocated. The compiler only uses two buffers. The operations recognized by Pidgin are limited to read and write, but there are useful extensions (append, read or write). The block number is the key to a rudimentary file system as described in *DDJ* #56. This simple file system will allow programs to be written in Pidgin that give named files and other nice features. In this way a rather machine-independent operating system is possible. The block number is a number from one through some maximum that specifies a unique external place to put the data contents of one buffer. This avoids dependence on a particular machine by pushing the translation from block number to drive, track, sector, or what have you, down to assembly-language support programs. The translation is responsible for setting a reasonable sequence for speed of access. But all that is required is a unique map between disk sectors (or whatever) and block number.

The file input-output statements all return an error code in the specially designated BYTE variable ER. Error code zero indicates all normal; one indicates end of file or end of medium; two indicates illegal operation; and three indicates all other problems.

OPEN returns error code two if the buffer was already open, zero otherwise. This indicates that the "buffer" will contain at least some information beyond the file contents. (The Apple requires a 17-byte table to control disk input-output, which is included in the space allocated for the buffer, for instance.) For this reason, the declared lengths of the buffers will be machine dependent.

CLOSE $$ specifies a buffer. It returns ER=2 if the buffer was already closed. It marks the buffer closed. If the buffer was open to write, then CLOSE will flush the buffer, writing the last block of data, and making any closed file marks.

READ $$ FROM $$ specifies a BYTE variable with the first two parameters and a buffer with the last two. This statement reads one character from the specified buffer until it has read all of the characters in the buffer. Then it refills the buffer by getting the next sequential block number of data from the external medium. When end of file is reached, it returns both ER=1 and the character 0–1.

WRITE $$ INTO $$ is the converse. It will write one specified character into the specified buffer until the buffer is full. It then writes the buffer onto the external medium and prepares to receive more data from the program. It keeps track of the block number to write on next, writing on se-

quentially incremented blocks. It returns ER=1 if the block number is greater than the maximum allowed, or zero.

READBUF $$ and WRITEBUF $$ give an access to the low-level programs that read and write one buffer of data from the external medium. They will be useful for programs that implement directory files or random-access files, but they are not used in the compiler. Programming the subroutines to support these file input-output statements took a substantial fraction of the time to make the compiler, perhaps a third.

The last "statement" in the macros is the null macro. It allows any line started with the end-line symbol to be ignored; that is, it allows line-long comments.

Implementing Tincmp, a compiler for Pidgin

In hopes of getting back a return of portable programs, I offer a compiler to translate Pidgin to 6502. The compiler is in two parts, a macro processor (Tincmp) written in Pidgin, and two sets of macros. I term these tools a compiler because they can generate machine code from Pidgin language. Others might feel that with no more error checking than it has, it couldn't be so dignified.

The value of the Pidgin listing of the Tincmp macro processor when you don't yet have a Pidgin compiler is this: you can easily translate Pidgin to your local Basic. If you translate Tincmp to Basic, then you can use the macros provided to form a compiler from Pidgin to 6502 in Basic. Feed that compiler the Pidgin code for Tincmp and you will have a compiler from Pidgin to 6502 in 6502. It runs much faster. This is a reasonable way to get started because the macros are the hard part, and can be used with two versions of the macro processor. I have written several versions of Tincmp, and find it takes a few days to get the macro processor working. The macros on the other hand took several weekends. When I add macros, I find I can add about 20 in a weekend.

So to implement this Pidgin compiler you need to translate Tincmp to some language you now have. To help in that let me first explain what Tincmp does, then how it does it.

Tincmp has two inputs – a set of *macros*, and an input text. Each macro consists of a *template* and a *replacement*. Templates start with a special *begin-template* character and end with the special *end-line* character. Between these two characters there can be up to nine *parameter flags* and any number of other characters besides these three or newline. The end-line character can be followed by any characters at all, then a newline. All the characters after the end-line are ignored, so they can and should be used for comments.

To summarize, a template consists of
1. begin-template character
2. zero or more ordinary characters or zero through nine parameter flags in any order
3. end-line character
4. zero or more comment characters
5. newline

If the end-line character is omitted, it will be assumed at the first newline. The macros attached show many examples using ':' as the begin-template character, ';' as the end-line character and '$' as the parameter flag.

The replacement consists of zero or more lines. Each replacement line may have
1. zero or more ordinary characters or "operation codes"
2. an optional end-line character followed by zero or more comment characters
3. newline

The power of Tincmp lies in its operation codes. It can be thought of an as interpreter for a very simple machine language. The simplicity is that there are no branching or repetition codes. The operation codes basically move data between ten registers, a stack, and the output.

The first line of the macros gives copies of the changeable special characters. If the first character is 'X', no newlines will be put into the output. This is required to generate code at times, and text at other times. The second character is a copy of the begin-template character. The third character is a copy of the end-line character. The fourth character is a copy of the parameter flag. The fifth is a copy of the operation code character. The sixth is an "ignore" character (set to tab, but invisible in the listings). The ignore character is ignored wherever it occurs in the macros. Its purpose is to help make the macros more readable. The first line must consist of exactly six characters and a newline. Otherwise an error is noted, and processing stops. With all these special characters, a way around them when necessary is available. The character '@' is a (fixed) escape character. Any one character (including newline and @) following the @ will be accepted as part of the macros. You will see it used heavily in the second macro set, because the definitions of the labels created might be special characters, just by chance.

An important link between the template and the operation codes takes place while Tincmp is matching the templates

Table 1

Fetch Operations

P Parameter	Fetch the value of the parameter specified by the index code.
V conVert	Fetch the value of the parameter specified by the index code. Subtract the character code for zero. If the result is not between zero and nine, set it to zero.
L Literal	Fetch the index code as a character.
N Number	Fetch the index code converted as a digit.
! pop	Fetch the top of the stack, decrementing the stack pointer.
S Stack	Fetch the top of the stack.
U Unique	Fetch a unique number. The numbers start at 100, and increase sequentially. (Default).
H Hex	Fetch the index code and the dispose code, interpreted as hexadecimal digits. Set the dispose code to 'C'.
T Trace	Set the trace mode on for the remainder of the macro. Skip the dispose section.

against input text. A line of input text *matches* a template if it is the same length as the template and if each ordinary character in the template is matched exactly. Any single character matches a parameter flag AND the character matched by the n'th parameter flag is placed in the n'th register, n=1, . . .,9. Thus the operation codes can manipulate the parameters found.

When a template has been matched, ordinary characters in the replacement are output without change. The end-line symbol causes the output of a newline. Any comments and the newline are ignored. Each operation code consists of four characters, the operation code flag, the fetch code, the index code, and the dispose code. Table 1 gives the fetch codes and Table 2 gives the dispose codes. The index code may modify the fetch code or the dispose code (or both). Let me comment on them briefly.

The value placed in the register is the internal machine representation of the character matched. The P fetch gets this value. If some other number has been stored in a register since the matching, of course P will now get that number. The registers hold integer numbers. If the parameter should be a digit, the V fetch gets a number from zero through nine. Notice that any non-digit is coerced to zero. The register is selected for these two fetches by the index code, and register zero can be selected. Since it cannot be disturbed by parameters, it can be used for a counter. The L and N fetches get data from the index code. L gets the internal machine representation of any character while N coerces it to a digit. The number on top of the stack can be fetched either with (!) or without (S) popping the stack. The fetch code H interprets both the index code and the dispose code and the hexadecimal representation of a byte and outputs it. The fetch code U allows generating ten symbols unique to a given macro. Tincmp keeps a symbol counter which starts at 100. During a macro, the U fetch adds the index code to the current symbol counter to return a symbol. Only at the end of the macro is the symbol counter increased, and then by the maximum index used during the macro, plus one. This means that references to symbol '0' during one macro will return the same number. It is intended to be used for generating labels.

Tincmp requires three parameters to be provided in IAV, the global vector mentioned above in describing BEGINMAIN. These give the starting blocks for the files to hold (1) the macros, (2) the input text, (3) the output text.

That's what Tincmp does. This is how it does it: the subroutine IN reads the flag line and initializes constants and variables. The subroutine RM reads the macro file, loading an array with information. The information is stored as shown in Table 3. After reading the macros, the input is read. Each line is compared to each template in turn until a match is found, or until all templates have been checked. If no match is found, the line is written out unchanged. If a match is found, then the subroutine DM interprets the replacement text. When an operation code is found, the next three characters are picked up. A branch on the first character does the fetch and a branch on the third does the dispose.

Tincmp can be used in two functionally different ways to compile Pidgin. One way is to use it as a preprocessor to generate code for an assembler. The other way is to run it three times and let it do its own assembly. The first way may have the advantage that an assembler will check the code for missing variables, twice-defined labels, etc. It also leads to simpler macros. However, I found when I tried it with the SCII assembler I use, that it was slow to transfer the output file from Tincmp into the assembler. Also, even as small a program as Tincmp expanded into 23K bytes of standard assembler code.

Table 2

Dispose Codes

P Parameter	Set the value of the parameter specified by the index code to the fetched.
S Stack	Increment the stack pointer and store the fetched on the stack.
C Character	Write the low byte of the fetched.
H High	Write the high byte of the fetched.
N Number	Write the fetched in decimal characters without leading zeros. (Default.)
+ add	Add the fetched to the top of the stack.
– subtract	Subtract the fetched from the top of the stack.
*** multiply**	Multiply the top of the stack by ten, then add the fetched.

Table 3

Storage of Macros

```
storage of
    :T1;COMMENTS
    R1;ALL
    :T2$;ARE
    R2 PIC; IGNORED
```

index	ILP	index	LS
0	0	0	T
1	6	1	1
2	17	2	;
. . .		3	R
		4	1
		5	;
		6	T
		7	2
		8	$
		9	;
		10	R
		11	2
		12	
		13	P
		14	1
		15	C
		16	;
		. . .	

The assembler swallowed this (slowly), but a slightly larger file would have to be cut into chunks, which would be quite cumbersome. This was too bad, because once inside, it assembled in about ten seconds.

The other way requires macros that are difficult to understand, but it gets the job done faster for me. The first set of macros produces an intermediate file in which all 6502 instructions have been coded, but with all operands left on single lines following the instruction codes. The first set of macros is thus rather like machine code. The second set of macros produces a file which is a third set of macros. This set of macros is simpler than the first, because its most frequent task is just to count the program bytes. The third set of macros contains the definitions for all labels and variables. When the third set of macros is used on the intermediate file, the completed 6502 code is output. I have included macros for this method since I have found it more useful.

Using the macros and Tincmp shown here, Tincmp compiles itself in 2 minutes 10 seconds on my Apple II. This is 50, 35, and 45 seconds for the three passes. The disk is continuously on during this time, so I believe the processing time roughly balances the input/output time. When compiled, the code occupies 3.5K bytes. It uses 4.7K bytes to store the first and larger macro set, and another thousand or so bytes for buffers, variables, etc.

In making these macros I originally tried a method I've since abandoned, and about which you might profit by a warning. One of the problems with having two sets of macros is to keep the same targets in each, with compatible definitions. I thought it might be a good idea to start from a single file and produce these two sets of macros from it by using two other sets of macros. It worked, but had two problems worse than the one it solved. First, it was confusing keeping several different sets of special characters in mind. Second, I kept having to change two sets of macros anyhow, because another macro would usually bring in a new concept and therefore a new command. The macros shown here evolved from such a process, however, and have some traces of that mechanical production remaining.

In my view macros should use every byte-saving or microsecond-saving trick known to programmers. When faced with a choice of time or space, I've usually chosen the faster, more space-consuming macro. If you can code any of the Pidgin statements with a saving in either space or time and not worsen the other, I would be glad to hear about it.

Genealogy

Tincmp's immediate parent is SIMCMP, described by W. Waite in *Implementing Software for Non-Numeric Applications*. SIMCMP is even simpler than Tincmp, having in essence exactly three kind of operation codes (UnN, PnC, VnN). With these much simpler codes, Waite defines the First Language Under Bootstrap (FLUB), a simpler language than Pidgin, having about 30 statements. I liked the approach of SIMCMP very much, but FLUB was very hard to write, and it seemed that I would need to write a variety of programs in the first language before I could move on to the second. (Like an editor, and some kind of executive.) Pidgin has proved convenient enough that I haven't yet felt like I had to get a better language going, but maybe soon.

Availability in Machine-Readable Form

The program and macros published here and last month give you a long start in implementing Pidgin on your system. If you want to save yourself some typing, and can transfer files from an Apple near you, you may be interested in the following offer. An experimental operating system written in Pidgin is available from the author for software experimenters. It includes Pidgin and 6502 versions of Tincmp, an editor, a tiny executive, some utilities, and Pidgin test programs. It can now be run on Apple, and can be configured for 1 through 4 floppy disk drives, using 13- or 16-sector disks. It requires a 16K memory board or an Apple II (not plus). Two disks and documentation are available for $20. A third disk with Meta4, the compiler generator to be described next month, is available for an additional $10. If you are reading this after 1981, better write to check availability. You can write me at 439 S. Orange Ave., S. Orange, NJ 07079. ■■

LISTING ON PAGE 24

Pidgin (Text on pages 10-14)

First Macro Set

```
U:;S^
:BYTE $$;
        ^P0S^N1+^!0P
:INT I$$;
        ^P9S^N2-^!9P
:BYTE $$($$$);
        ^P9S^V3S^V4*^V5*^!0-^!9P
        ^HA9;
        ^P9C;
        ^H85^P0C;
        ^P0S^N1+^!0P
        ^HA9;
        ^P9H;
        ^H85^P0C;
        ^P0S^N1+^!0P
:INT I$$($$$);
        ^P9S
        ^V3S^V4*^V5*^S0+^!0-
        ^!9P
        ^HA9;
        ^P9C;
        ^H85^P0C;
        ^P0S^N1+^!0P
        ^HA9;
        ^P9H;
        ^H85^P0C;
        ^P0S^N1+^!0P
:BYTE $$($$$$$);
        ^P9S
        ^V3S^V4*^V5*^V6*^V7*^!0-
        ^!9P
:INT I$$($$$$$);
        ^P9S
        ^V3S^V4*^V5*^V6*^V7*^S0+^!0-
        ^!9P
:$$='$';
        ^HA9;
        ^P3C;
        ^H85;
        ^P1C^P2C;
:$$=+$$$;
        ^HA9;
        ^V3S^V4*^V5*^!0C;
        ^H85;
        ^P1C^P2C;
:I$$=+$$$$$;
        ^HA9;

        ^V3S^V4*^V5*^V6*^V7*^S0C;
        ^H8D;
        LI^P1C^P2C;
        ^HA9;
        ^!0H;
        ^H8D;
        HI^P1C^P2C;
:$$=$$;
        ^HA5;
        LI^P5C^P6C;
        ^HB9;
        L^P3C^P4C;
        ^H0A;
        ^H85;
        ^P1C^P2C;

:I$$=I$$(I$$);
        ^HAD;
        LI^P5C^P6C;
        ^H0A;
        ^HAA;
        ^HAD;
        HI^P5C^P6C;
        ^H2A;
        ^HA8;
        ^H8A;
        ^H18;
        ^H69;
        LI^P3C^P4C;
        ^H85;
        LBASE;
        ^H98;
        ^H69;
        HI^P3C^P4C;
        ^H85;
        HBASE;
        ^HA0;
        ^H01;
        ^HB1;
        LBASE;
        ^H99;
        LI^P1C^P2C;
        ^H88;
        ^HF0;
        ^N0S^N8-^!0C;

:$$($$)=$$;
        ^HA4;
        ^P3C^P4C;
        ^HA5;
        ^P5C^P6C;
        ^H91;
        ^P1C^P2C;

:I$$($$)=I$$;
        ^HA5;
        ^P3C^P4C;
        ^H0A;
        ^HA8;
        ^HAD;
        LI^P5C^P6C;
        ^H91;
        I^P1C^P2C;
        ^HC8;
        ^HAD;
        HI^P5C^P6C;
        ^H91;
        I^P1C^P2C;

        ^HE5;
        ^P5C^P6C;
        ^H85;
        ^P1C^P2C;
:$$++;
        ^HE6;
        ^P1C^P2C;
:$$--;
        ^HC6;
        ^P1C^P2C;
:I$$++;
        ^HEE;
        LI^P1C^P2C;
        ^HD0;
        ^H03;
        ^HEE;
        HI^P1C^P2C;

:I$$--;
        ^HA9;
        ^H00;
        ^HCD;
        LI^P1C^P2C;
        ^HD0;
        ^H03;
        ^HCE;
        HI^P1C^P2C;
        ^HCE;
        LI^P1C^P2C;

:I$$=-I$$;
        ^HA9;
        ^H00;
        ^H38;
        ^HED;
        LI^P3C^P4C;
        ^H8D;
        LI^P1C^P2C;
        ^HA9;
        ^H00;
        ^HED;
        HI^P3C^P4C;
        ^H8D;
        HI^P1C^P2C;

:I$$=I$$+I$$;
        ^HAD;
        LI^P3C^P4C;
        ^H18;
        ^H6D;
        LI^P5C^P6C;
        ^H8D;
        LI^P1C^P2C;
        ^HAD;
        HI^P3C^P4C;
        ^H6D;
        HI^P5C^P6C;
        ^H8D;
        HI^P1C^P2C;

        ^HA5;
        ^P3C^P4C;
        ^HC5;
        ^P5C^P6C;
        ^HF0;
        ^H02;
        ^HA2;
        ^HFF;
        ^H86;
        ^P1C^P2C;

:$$=$$==$$;
        ^HA2;
        ^HFF;
        ^HA5;
        ^P3C^P4C;
        ^HC5;
        ^P5C^P6C;
        ^HF0;
        ^H02;
        ^HA2;
        ^H00;
        ^H86;
        ^P1C^P2C;

:$$=$$<!$$;
        ^HA2;
        ^HFF;
        ^HA5;
        ^P3C^P4C;
        ^HC5;
        ^P5C^P6C;
        ^H90;
        ^H02;
        ^HA2;
        ^H00;
        ^H86;
        ^P1C^P2C;

:$$=$$<=$$;
        ^HA2;
        ^H00;
        ^HA5;
        ^P5C^P6C;
        ^HC5;
        ^P3C^P4C;
        ^H90;
        ^H02;
        ^HA2;
        ^HFF;
        ^H86;
        ^P1C^P2C;
:$$=I$$!=I$$;
        ^HA2;
        ^H00;
        ^HA0;
        ^H01;
        ^HB9;
        LI^P3C^P4C;
```

```
        ^P3C^P4C;
        ^H85;
        ^P1C^P2C;
:I$$=$$;
        ^HA5;
        ^P3C^P4C;
        ^H8D;
        LI^P1C^P2C;
        ^HA9;
        ^H00;
        ^H8D;
        HI^P1C^P2C;
:$$=I$$;
        ^HAD;
        LI^P3C^P4C;
        ^H85;
        ^P1C^P2C;
:I$$=I$$;
        ^HAD;
        LI^P3C^P4C;
        ^H8D;
        LI^P1C^P2C;
        ^HAD;
        HI^P3C^P4C;
        ^H8D;
        HI^P1C^P2C;
:$$=$$($$);
        ^HA4;
        ^P5C^P6C;
        ^HB1;
        ^P3C^P4C;
        ^H85;
        ^P1C^P2C;
:I$$=I$$($$);
        ^HA5;
        ^P5C^P6C;
        ^H0A;
        ^HA8;
        ^HB1;
        I^P3C^P4C;
        ^H8D;
        LI^P1C^P2C;
        ^HC8;
        ^HB1;
        I^P3C^P4C;
        ^H8D;
        HI^P1C^P2C;

:$$=$$(I$$);
        ^HA9;
        H^P3C^P4C;
        ^H18;
        ^H6D;
        HI^P5C^P6C;
        ^H9D;
        L^U0N;
        ^HAC;
```

(Continued on page 24, column 2)

```
:$$(I$$)=$$;
        ^HA9;
        H^P1C^P2C;
        ^H18;
        ^H6D;
        HI^P3C^P4C;

        ^H8D;
        L^U0N;
        ^HAC;
        LI^P3C^P4C;
        ^HA5;
        ^P5C^P6C;
        ^H99;
        L^P1C^P2C;
        ^H0A;
:I$$(I$$)=I$$;
        ^HAD;
        LI^P3C^P4C;
        ^H0A;
        ^HAA;
        ^HAD;
        HI^P3C^P4C;
        ^H2A;
        ^HA8;
        ^H8A;
        ^H18;
        ^H69;
        LI^P1C^P2C;
        ^H85;
        LBASE;
        ^H98;
        ^H69;
        HI^P1C^P2C;
        ^H85;
        HBASE;

        ^HA0;
        ^H01;
        ^HB9;
        LI^P5C^P6C;
        ^H91;
        LBASE;
        ^H88;

        ^HF0;
        ^N0S^N8-^!0C;
:$$=$$+$$;
        ^HA5;
        ^P3C^P4C;
        ^H18;
        ^H65;
        ^P5C^P6C;
        ^H85;
        ^P1C^P2C;
:$$=$$-$$;
        ^HA5;
        ^P3C^P4C;
        ^H38;
```

(Continued on page 24, column 3)

```
:I$$=I$$-I$$;
        ^HAD;
        LI^P3C^P4C;
        ^H38;
        ^HED;
        LI^P5C^P6C;
        ^H8D;
        LI^P1C^P2C;
        ^HAD;
        HI^P3C^P4C;
        ^HED;
        HI^P5C^P6C;
        ^H8D;
        HI^P1C^P2C;
:I$$=I$$*I$$;
        ^H20;
        IMUL;


        LI^P3C^P4C;
        LI^P5C^P6C;
        ^H8D;
        LI^P1C^P2C;
        ^H8C;
        HI^P1C^P2C;
:I$$=I$$/I$$;
        ^H20;
        IDIV;
        LI^P3C^P4C;
        LI^P5C^P6C;


        ^H8D;
        LI^P1C^P2C;
        ^H8C;
        HI^P1C^P2C;
:$$=$$&$$;
        ^HA5;
        ^P3C^P4C;
        ^H25;
        ^P5C^P6C;
        ^H85;
        ^P1C^P2C;
:$$=$$?$$;
        ^HA5;
        ^P3C^P4C;

        ^H05;
        ^P5C^P6C;
        ^H85;
        ^P1C^P2C;
:$$=!$$;
        ^HA5;
        ^P3C^P4C;
        ^H49;
        ^HFF;
        ^H85;
        ^P1C^P2C;
:$$=$$!=$$;
        ^HA2;
        ^H00;
```

(Continued on page 24, column 4)

```
        ^HD9;
        LI^P5C^P6C;
        ^HF0;
        ^H02;
        ^HA2;
        ^HFF;
        ^H88;
        ^H10;
        ^N0S^N1S^N3*^!0-^!0C;
        ^H86;

        ^P1C^P2C;
:$$=I$$==I$$;
        ^HA2;
        ^HFF;
        ^HA0;
        ^H01;
        ^HB9;
        LI^P3C^P4C;
        ^HD9;
        LI^P5C^P6C;
        ^HF0;
        ^H02;
        ^HA2;

        ^H00;
        ^H88;
        ^H10;
        ^N0S^N1S^N3*^!0-^!0C;
        ^H86;
        ^P1C^P2C;
:$$=I$$<!I$$;
        ^HA2;
        ^HFF;
        ^HAD;
        HI^P3C^P4C;
        ^HCD;
        HI^P5C^P6C;


        ^H4D;
        HI^P5C^P6C;
        ^HF0;
        ^H09;
        ^H10;
        ^H0D;
        ^HAD;
        HI^P5C^P6C;
        ^H10;
        ^H0A;
        ^H30;
        ^H08;
        ^HAD;
        LI^P3C^P4C;
        ^HCD;
        LI^P5C^PC6;
        ^H90;
        ^H02;
        ^HA2;
        ^H00;
```

(Continued on next page)

Pidgin (Listing continued, text on pages 10-14)

```
        ^H86;
        ^P1C^P2C;
:$$=I$$<=I$$;
        ^HA2;
        ^H00;
        ^HAD;


        HI^P5C^P6C;
        ^HCD;
        HI^P3C^P4C;
        ^H4D;
        ^HI^P3C^P4C;
        ^HF0;
        ^H09;
        ^H10;
        ^H0D;
        ^HAD;
        HI^P3C^P4C;
        ^H10;
        ^H0A;
        ^H30;
        ^H08;
        ^HAD;
        LI^P5C^P6C;
        ^HCD;
        LI^P3C^P4C;
        ^H90;
        ^H02;
        ^HA2;
        ^HFF;
        ^H86;
        ^P1C^P2C;


:SUB $$;
:GOSUB $$;
        ^H20;
        S^P1C^P2C;
:CALL I$$;
        ^HAD;
        LI^P1C^P2C;
        ^H85;
        LBASE;
        ^HAD;
        HI^P1C^P2C;
        ^H85;
        HBASE;
        ^H20^H65^H96;
:RETURN;
        ^H60;
:ENDSUB;
        ^H60;
:GOTO $$;
        ^H4C;
        U^P1C^P2C;
:LOC $$;
:STOP $;
        ^HA9;
        ^P1C;
        C^S0N;
:CASE $$;
        ^!0P
        ^H4C;
        C^S0N;
        ^HA5;
        ^P1C^P2C;
        ^HC5;
        SWT;
        ^HF0;
        ^H03;
        ^U0S
        ^H4C;
        C^S0N;


:DEFAULT;
        ^!0P
        ^H4C;
        C^S0N;
        ^U0S
:ENDCHOOSE;
        ^!0P
        ^!0P
:MS '$$$$$$$$$';
        ^H20;
        IMSG;
        ^P1C^P2C^P
        ^P4C^P5C^P
        ^P7C^P8C^P9C;
:WRITE $$ INTO $$;
        ^HA9;
        L^P3C^P4C;
        ^HA0;
        H^P3C^P4C;
        ^HA6;
        ^P1C^P2C;


        ^H20;
        PTCH;
:WRITE $$;
        ^HA5;
        ^P1C^P2C;
        ^H20;
        WRIT;
:WRITEBF $$;
        ^HA9;
        L^P1C^P2C;
        ^HA0;
        H^P1C^P2C;
        ^H20;
        BFRF;
        ^H20;
        WRBF;
:READBF $$;
        ^HA0;
        L^P1C^P2C;
        ^HA0;
        H^P1C^P2C;
        ^HA5;
        ^P3C^P4C;
        ^H8D;
        HI^P1C^P2C;
:;
```

Second Macro Set

```
U:;$^
:BYTE $$;
        )^P1C^P2C(;
        @@^P0C;
        ^P0S^N1+^!0P
:INT I$$;
        ^P9S^N2-^!9P
        )LI^P1C^P2C(;
        @@^P9C@@^P9H;
        )HI^P1C^P2C(;
        ^P9S^N1+@@^S0C@@^!0H;


:BYTE $$($$$);
        )^P1C^P2C(;
        @@^P0C;
        ^P9S
        ^V3S^V4*^V5*^!0-^!9P
        ^N4+
        ^P0S^N1+^!0P
        ^N4+
        ^P0S^N1+^!0P



:INT I$$($$$);
        )I^P1C^P2C(;
        @@^P0C;
        ^P9S
        ^V3S^V4*^V5*^S0+^!0-
        ^!9P
        ^N4+
        ^P0S^N1+^!0P
        ^N4+
        ^P0S^N1+^!0P



:BYTE $$($$$$$);
        ^P9S
        ^V3S^V4*^V5*^V6*^V7*^!0-
        ^!9P
        )L^P1C^P2C(;
        @@^P9C;
        )H^P1C^P2C(;
        @@^P9H;


:INT I$$($$$$$);
        ^P9S
        ^V3S^V4*^V5*^V6*^V7*^S0+^!0-
        ^!9P
:$$=$$+$$;
        ^N7+
:$$=$$-$$;
        ^N7+
:$$++;
        ^N2+
:$$--;
        ^N2+


:I$$++;
        ^N8+
:I$$--;
        ^N9+
        ^N4+
:I$$=-I$$;
        ^N9+
        ^N8+
:I$$=I$$+I$$;
        ^N1+
        ^N9+
        ^N9+
:I$$=I$$-I$$;
        ^N1+
        ^N9+
        ^N9+
:I$$=I$$*I$$;
        ^N9+
        ^N4+
:I$$=I$$/I$$;
        ^N9+
        ^N4+


:$$=$$&$$;
        ^N6+
:$$=$$?$$;
        ^N6+
:$$=!$$;
        ^N6+
:$$=$$!=$$;
        ^N8+
        ^N4+
:$$=$$==$$;
        ^N6+
        ^N6+
:$$=$$<!$$;
        ^N8+
        ^N4+
:$$=$$<=$$;
        ^N6+
        ^N6+
:$$=I$$!=I$$;
        ^N9+
        ^N9+
        ^N1+
:$$=I$$==I$$;
        ^N9+
        ^N9+
        ^N1+
```

```
        ^HAE;
        SPSV;
        ^H9A;
        ^H60;
:TOP;
:HIMEM=$$$$$;

        ^V1S^V2*^V3*^V4*^V5*^!9P
:LOMEM=$$$$$;
:REGISTER=$$$;
        ^V1S^V2*^V3*^!0P
:BOTTOM;
        ^HFF;
:BEGINMAIN(AC,IAV);
        ^HBA;
        ^H8E;
        SPSV;
:ENDMAIN;
        ^H60;
:IF $$;
        ^HA5;
        ^P1C^P2C;
        ^HD0;
        ^H03;
        ^U0S
        ^H4C;
        I^S0N;
:ENDIF;
        ^!0P
:ELSE;
        ^!0P
        ^U0S

        ^H4C;
        I^S0N;
:WHILE;
        ^U0S
:ON $$;
        ^HA5;
        ^P1C^P2C;
        ^HD0;
        ^H03;
        ^U0S
        ^H4C;
        W^S0N;
:ENDWHILE;
        ^!0P
        ^H4C;
        W^S0N;
        ^!0P
:CHOOSE ON $$;
        ^HA5;
        ^P1C^P2C;
        ^H85;
        SWT;
        ^U0S
        ^U1S
        ^H4C;
```

(Continued on page 26, column 2)

```
        ^H20;
        BFRF;
        ^H20;
        RDBF;
:READ $$ FROM $$;
        ^HA9;
        L^P3C^P4C;
        ^HA0;
        H^P3C^P4C;
        ^H20;
        GTCH;
        ^H85;
        ^P1C^P2C;
:READ $$;


        ^H20;
        READ;
        ^H85;
        ^P1C^P2C;
:OPEN $$ FOR $$ AT I$$;
        ^HAC;
        HI^P5C^P6C;
        ^HAD;
        LI^P5C^P6C;
        ^H20;
        BDTS;
        ^HA9;
        L^P1C^P2C;
        ^HA0;
        H^P1C^P2C;
        ^H20;
        BFRF;
        ^HA5;
        ^P3C^P4C;
        ^H20;
        OPEN;


:CLOSE $$;
        ^HA9;
        L^P1C^P2C;
        ^HA0;
        H^P1C^P2C;
        ^H20;
        CLOS;
:UNPACK(I$$,$$,$$);
        ^HAD;
        HI^P1C^P2C;
        ^H85;
        ^P3C^P4C;
        ^HAD;
        LI^P1C^P2C;
        ^H85;
        ^P5C^P6C;
:PACK(I$$,$$,$$);
        ^HA5;
        ^P5C^P6C;
        ^H8D;
        LI^P1C^P2C;
```

(Continued on page 26, column 3)

```
        )LI^P1C^P2C(;
        @@^P9C;
        )HI^P1C^P2C(;
        @@^P9H;



:$$='S';
        ^N4+
:$$=+$$$;
        ^N4+
:I$$=+$$$$$;
        ^N7+
        ^N3+
:$$=$$;
        ^N4+
:I$$=$$;
        ^N7+
        ^N3+
:$$=I$$;
        ^N5+
:I$$=I$$;
        ^N9+
        ^N3+
:$$=$$($$);
        ^N6+
:I$$=I$$($$);
        ^N9+

        ^N6+
:$$=$$(I$$);
        ^N9+
        ^N6+
        )L^U0N(;
        ^S0S^N1-@@^S0C@@^!0H;
        ^N2+
:I$$=I$$(I$$);
        ^N9+



        ^N7+
        ^N9+
        ^N6+
:$$($$)=$$;
        ^N6+
:I$$($$)=I$$;
        ^N7+
        ^N8+
:$$(I$$)=$$;
        ^N9+
        ^N8+
        )L^U0N(;
        ^S0S^N1-@@^S0C@@^!0H;
:I$$(I$$)=I$$;
        ^N9+
        ^N8+
        ^N9+
        ^N5+
```

(Continued on page 26, column 4)

```
:$$=I$$<!I$$;
        ^N9+
        ^N9+
        ^N9+
        ^N7+
:$$=I$$<=I$$;
        ^N9+
        ^N9+
        ^N9+
        ^N7+
:SUB $$;
        )S^P1C^P2C(;
        @@^S0C@@^S0H;


:GOSUB $$;
        ^N3+
:CALL I$$;
        ^N9+
        ^N4+

:RETURN;
        ^N1+
:ENDSUB;
        ^N1+
:GOTO $$;
        ^N3+
:LOC $$;
        )U^P1C^P2C(;
        @@^SOC@@^S0H;
:STOP $;
        ^N7+

:TOP;


        X)(SXX;
        )SWT(;
        @@^H0D;
        )LBASE(;
        @@^H0E;
        )HBASE(;
        @@^H0F;
        )ER(;
        @@^H0B;
        )PAG(;
        @@^H03;
        )COL(;
        @@^H05;
        )IMSG(;
```

(Continued on next page)

Pidgin (Listing continued, text on pages 10-14)

```
        @@^H00@@^H90;
        )IMUL(;
        @@^H55@@^H90;
        )IDIV(;
        @@^H2A@@^H90;
        )BDTS(;
        @@^HBF@@^H90;
        )BFRF(;
        @@^HC2@@^H90;
        )OPEN(;
        @@^HC5@@^H90;
        )CLOS(;
        @@^HC8@@^H90;
        )RDBF(;
        @@^HCB@@^H90;
        )GTCH(;
        @@^HCE@@^H90;
        )WRBF(;
        @@^HD1@@^H90;
        )PTCH(;
        @@^HD4@@^H90;
        )READ(;
        @@^HD7@@^H90;
        )WRIT(;
        @@^HDA@@^H90;


:HIMEM=$$$$$;
        ^V1S^V2*^V3*^V4*^V5*^!9P
:LOMEM=$$$$$;
        ^V1S^V2*^V3*^V4*^V5*
:REGISTER=$$$;
        ^V1S^V2*^V3*^!0P
:BOTTOM;
:BEGINMAIN(AC,IAV);
        )AC(;
        @@^H0C;
        )IAV(;
        @@^H09;
        ^P9S^N1-^!9P
        )SPSV(;
        @@^P9C@@^P9H;
        ^N4+


:ENDMAIN;
        ^N1+
:IF $$;
        ^N4+
        ^!8P^U0S^P8S
        ^N3+
:ENDIF;
        ^!9P)I^S0N(^P9S;
        @@^S0C@@^S0H;
        ^!8P^!9P^P8S
:ELSE;
        ^!8P^!9P^P8S
        ^N3+
        )C^P9N(;
        @@^S0C@@^S0H;
        ^N6+
        ^!8P^U0S^P8S
        ^N3+
:DEFAULT;
        ^!8P^!9P^P8S
        ^N3+
        )C^P9N(;
        @@^S0C@@^S0H;


        ^!8P^U0S^P8S
:ENDCHOOSE;
        ^!8P^!9P^P8S
        ^!9P)C^S0N(^P9S;
        @@^S0C@@^S0H;
        ^!8P^!9P^P8S
:MS '$$$$$$$$$';
        ^N3+
        ^N9+
:WRITE $$ INTO $$;
        ^N9+
:WRITE $$;
        ^N5+
:WRITEBF $$;
        ^N5+
        ^N5+
:READBF $$;
        ^N5+
        ^N5+
:READ $$ FROM $$;
        ^N9+
:READ $$;
        ^N5+
:OPEN $$ FOR $$ AT I$$;
        ^N3+
        ^N9+
        ^N9+
:CLOSE $$;
        ^N7+
:UNPACK(I$$,$$,$$);
        ^N5+
        ^N5+
:PACK(I$$,$$,$$);
        ^N5+
        ^N5+
:;
```

End Macro Set Listings

```
BYTE HF;'F'
BYTE LE;END OF LIST
BYTE LS(09000);LIST OF MACRO DEFINITIONS
BYTE MF;MACRO REPLACEMENT OPER FLAG
BYTE ML;MACRO LENGTH
BYTE MM;MINIMUM MACRO LEN

BYTE ND;NUMBER OF DIGITS- USED IN SUB SD FOR NUMBER OUTPUT
BYTE NL;NEWLINE
BYTE O1;FETCH CODE
BYTE O2;INDEX CODE
BYTE O3;DISPOSE CODE
BYTE OA;'+' ADD OPERATOR
BYTE OB;'!' POP STACK OPERATOR
BYTE OC;'C' CHARACTER DISPOSE
BYTE OD;'V' DIGIT CONVERSION FETCH
BYTE OE;ESCAPE CHARACTER
BYTE OG;IGNORE CHARACTER
BYTE OH;'H' HEX CONVERSION FETCH
BYTE OL;'L' LITERAL FETCH
BYTE OM;'*' MULTIPLY DISPOSE
BYTE ON;'N' NUMERIC LITERAL FETCH

BYTE OP;'P' PARAMETER FETCH OR DISPOSE
BYTE OR;'-' REDUCE (SUBTRACT) DISPOSE
BYTE OS;'S' STACK FETCH OR DISPOSE
BYTE OT;TRACE FLAG TURN ON
BYTE PP;POINTER INTO IPR
BYTE RB;BEGIN DEFINITION FLAG
BYTE RC;(COMMENT) END-LINE FLAG
BYTE SF;SUBSTITUTION (PARAMETER) FLAG
BYTE SP;STACK POINTER
BYTE TR;TRUE IF TO TRACE
BYTE UG;USE IGNORE; TRUE UNLESS OG IS 'X'

BYTE UN;NOT X-- FLAG FOR NOT SUPPRESSING NEWLINES ON OUTPUT
BYTE UO;USE OPERATIONS-- TRUE UNLESS MF IS 'X'
BYTE UT;USE TRACE  TRACE MODE IS ON
BYTE ZR;CHARACTER ZERO
INT I00;CONSTANT 0
INT I01;CONSTANT 1
INT I09;CONSTANT 9
INT I10;CONSTANT 10
INT I16;CONSTANT 16
INT IAA;WORK
INT IBB; WORKING STORAGE
INT IBC;BUCKET NUMBER
INT IDP;DEFINITION POINTER WHILE MATCH
INT IED;POINTS TO END OF DEFINITIONS
INT III; POINTER TO L WHILE READING
INT IJJ; POINTER TO L READING CODE
INT ILM;MAXIMUM LIMIT FOR STORING IN L

INT ILP(01000);POINTERS TO MACROS
INT IMP;MACRO POINTER DURING EXPANSION
INT INM;NUMBER OF MACROS
INT IPR(010);PARAMETER VALUES
```

```
        ^!8P^U0S^P8S
        ^N3+
        )I^P9N(;
        @@^S0C@@^S0H;
:WHILE;
        ^!8P^U0S^P8S
        ^!9P)W^S0N(^P9S;
        @@^S0C@@^S0H;
:ON $$;
        ^N4+
        ^!8P^U0S^P8S
        ^N3+
:ENDWHILE;
        ^!8P^!9P^P8S
        ^N3+
        )W^P9N(;
        @@^S0C@@^S0H;
        ^!8P^!9P^P8S
:CHOOSE ON $$;
        ^N4+
        ^!8P^U0S^P8S
        ^!9P^U1S^P9S
        ^N3+
:CASE $$;
        ^!8P^!9P^P8S
```

(Continued on page 28, column 2)

Tincmp Listing

```
TOP;TINCMP COPYRIGHT (C) 1981 W.A.GALE
LOMEM=16384;$4000
HIMEM=36860;$9000
REGISTER=086
BYTE AA;WORK
BYTE BB;WORK

BYTE BF(080);EXPANSION BUFFER
BYTE BL;BLANK
BYTE BP;POINTER INTO BF
BYTE C0;CONSTANT ZERO
BYTE C1;CONSTANT ONE
BYTE C2;CONSTANT TWO
BYTE C3;CONSTANT 3
BYTE C4;CONST 40
BYTE C8;CONSTANT EIGHTY
BYTE C9;CHARACTER! NINE
BYTE CC; INPUT CHARACTER
BYTE CX;CONSTANT TEN

BYTE DG;DIGIT FROM PARAMETER TREATMENT DEFINITION
BYTE DS(010);DIGIT STACK FOR SUB SD
BYTE EF;END OF FILE CHAR
BYTE F1(00275);INPUT BUFFER
BYTE F2(00275);OUTPUT BUFFER SIZES ARE MACHINE DEPENDENT
BYTE HA;'A'
```

(Continued on top of page 28)

```
INT ISS(040); INT TO HOLD NUMBERS-MAIN STACK
INT ITU;VALUE OF PARAMETER TO USE
INT IUU;SYMBOL GENERATOR(UNIQUE)
INT IXX;WORK
INT IYY;WORK
BEGINMAIN(AC,IAV)
NL=+141;!!!!!!!!!!!!!MACHINE DEPENDENT
MS 'COPYRIGHT'
MS ' (C) 1981'
MS ' W.A.GALE'
WRITE NL
GOSUB IN;INITIALIZE INCL READ FLAG LINE
GOSUB RM;READ MACROS INTO TABLE
LOC 00;EXPANSION
WHILE
        READ CC FROM F1
        AA=ER==C0
ON AA;THAT IS, UNTIL EOF IS REACHED ON INPUT

       IF UG
               WHILE
                       AA=CC==0G
               ON AA;IGNORE LEADING IGNORE CHARACTERS
                       READ CC FROM F1
               ENDWHILE
       ENDIF
       BP=C1;BUF POINTER
       BF(C0)=CC
       WHILE
               READ CC FROM F1
               AA=CC!=NL
               BB=BP!=C8
               AA=AA&BB
       ON AA;WHILE LESS THAN 80 CHAR AND NOT NEWLINE
               BF(BP)=CC;THEN PUT IT IN A BUFFER FOR MULTIPLE COMPARISON
               BP++


         ENDWHILE
         WHILE
         AA=CC!=NL
         ON AA


                 READ CC FROM F1
         ENDWHILE;HERE WE ARE DUMPING A LONG INPUT LINE
         BF(BP)=RC
         BP++
         BF(BP)=NL
         LE=BP
         AA=BP<=MM
         IF AA;TO SHORT TO MATCH
                 ML=+000
                 GOTO 17
         ELSE
                 ML=+001
         ENDIF
         IDP=I00
         PP=C0
         IJJ=I00
```

(Continued on next page)

Pidgin (Listing continued, text on pages 10-14)

```
        INM=C0
        WHILE
                AA=IDP<!IED;DEF PTR < END OF DEFINITIONS
        ON AA
                BP=C0
                WHILE
                        AA=BP<=LE
                ON AA
                        AA=LS(IJJ)
                        AA=AA==RC
                        03=BF(BP)
                        03=03==RC
                        AA=AA&03;IF BOTH THE BUFFER AND THE TARGET HAVE
;TEMPLATE END OF LINE, THEN WE HAVE MATCHED
                        IF AA
                                GOSUB DM;DO MACRO EXPANSION
                                GOTO 00
                        ELSE
                        AA=BF(BP)
                        BB=LS(IJJ)
                        AA=AA==BB
                        IF AA
                                GOTO 01;MATCHING
                        ELSE
                        AA=BB!=SF;NOT A TEMPLATE PARAMETER FLAG
                        IF AA
                                GOTO 10;MISMATCHED
                        ELSE;THIS IS A PARAMETER
                                PP++
                                AA=BF(BP)
                                IAA=AA
                                IPR(PP)=IAA
                        ENDIF
                        ENDIF
                        ENDIF
                LOC 01

                        BP++
                        IJJ++
                ENDWHILE
        LOC 10
                PP=C0
                INM++
                IDP=ILP(INM)
                IJJ=IDP
        ENDWHILE
        ;NO MATCH
LOC 17
        BP=C0
        WHILE
                CC=BF(BP)
                01=BP+C1
                AA=BF(01)
                AA=AA!=NL
        ON AA
                IF ML;THEN ALSO WRITE
                        WRITE CC
                ENDIF
                AA=I00<!IAA
ON AA
                ND--
                AA=DS(ND)
                WRITE AA
ENDWHILE
WRITE BL
ENDSUB


                                SUB CD;CONVERT AA AS DECIMAL DIGIT
BB=ZR<=AA
CC=AA<=C9
BB=BB&CC
IF BB


        AA=AA-ZR
        RETURN


ENDIF
AA=C0
ENDSUB

                                SUB CH;CONVERT AA AS HEX DIGIT
;ASSUMES THAT A-F ARE CONSECUTIVE

BB=ZR<=AA
CC=AA<=C9
BB=BB&CC
IF BB
        AA=AA-ZR
        RETURN
ENDIF
BB=HA<=AA
CC=AA<=HF
BB=BB&CC
IF BB
        AA=AA-HA
        AA=AA+CX


        RETURN
ENDIF
AA=C0
ENDSUB

                                SUB IN;INITIALIZE
ILM=+08920;DIMENSION OF LS -80
I00=+00000
I01=+00001
I10=+00010
I09=+00009
C0=+000
C1=+001
C2=+002
C3=+003
```

```
                WRITE CC INTO F2
                BP++
        ENDWHILE
        IF ML
                WRITE NL
        ENDIF
        IF UN;ONLY IF NOT SUPPRESSING
                WRITE NL INTO F2
        ENDIF
ENDWHILE
CLOSE F1
CLOSE F2
ENDMAIN
                SUB SD;THIS SUB CONVERTS "TOUSE" TO A NUMBER WITHOUT LEAD
AA=ITU<!I00
IF AA
        BB=+001
        ITU=-ITU
ELSE
        BB=+000
ENDIF
AA=ITU==I00
IF AA
        ND=C1
        DS(C0)=ZR
ELSE
        ND=C0
        WHILE
                AA=I00<!ITU
        ON AA
                IYY=ITU/I10
                IAA=I10*IYY
                IXX=ITU-IAA
                ITU=IYY
                AA=IXX
                AA=AA+ZR
                DS(ND)=AA
                ND++

        ENDWHILE
ENDIF
DS(ND)=OR
ND=ND+BB;IE INCR FOR NEG ONLY
ENDSUB
                        SUB WN;WRITE A NUMBER INTO F2
GOSUB SD;STACK THE DIGITS
WHILE;NOW WRITE OUT THE DIGITS FIRST TO LAST
        IAA=ND
        AA=I00<!IAA
ON AA
        ND--
        AA=DS(ND)
        WRITE AA INTO F2
ENDWHILE
ENDSUB
                        SUB PN;WRITE THE NUMBER ON THE TERMINAL
GOSUB SD;STACK THE DIGITS
WHILE
        IAA=ND
```

(Continued on top of page 30)

```
EF=+255;END OF FILE CHAR
C4=+040
C8=+080
I16=+00016
SP=+000
C9='9'
ZR='0'
BL=' ';BLANK
HF='F'
HA='A'
CX=+010

;READ CHARACTER FLAGS
IBC=IAV(C1)
TR='R'
CLOSE F1
OPEN F1 FOR TR AT IBC
IBC=IAV(C3)
TR='W'
CLOSE F2
OPEN F2 FOR TR AT IBC
READ AA FROM F1;X SUPPRESSES NEW LINE OUTPUT

OT='T'
UT=+000
BB='X'
UN=AA!=BB;UN SAYS THE CHAR WAS NOT X, SO DON'T SUPPRESS
READ RB FROM F1;BEGINNING OF DEFN FLAG
READ RC FROM F1;COMMENT AND END OF LINE FLAG
READ SF FROM F1;TEMPLATE PARAMETER FLAG
READ MF FROM F1;EXPANSION OPERATION FLAG

BB='X'
AA=MF==BB
IF AA
        UO=C0
ELSE
        UO=C1

ENDIF
OP='P';PARAMETER DESIGNATOR IN OPERATION SEQUENCE
OE='@';ESCAPE CHARACTER
OD='V';CONVERT PARAMETER TO DIGIT IN ACTION SEQUENCE
OB='!';POP STACK DESIGNATOR IN OPERATION SEQUENCE
OS='S';STACK DESIGNATOR IN OPERATION SEQUENCE
OH='H';HEX CONSTANT FETCH AND WRITE


;ALSO FOR HIGH BYTE OUTPUT
ON='N';LITERAL NUMERIC FETCH
OL='L';LITERAL BYTE FETCH
OC='C';CHARACTER OUT DESIGNATION
OA='+';ADD TO STACK DESIGNATION
OR='-';SUBTRACT (REDUCE) FROM STACK
OM='*'; MULTIPLY STACK BY BASE AND ADD
READ OG FROM F1;IGNORE CHARACTER
AA='X'
```

(Continued on next page)

Pidgin (Listing continued, text on pages 10-14)

```
BB=AA==OG
IF BB
        UG=+000
ELSE
        UG=+001
ENDIF
READ CC FROM F1;NEWLINE
AA=NL!=CC;NL IS NEWLINE
IF AA
        MS 'FLAG LINE'
        STOP 1
ENDIF
IUU=+00100
ENDSUB;IN
                        SUB RM;READ MACROS
III=I00
INM=C0
MM=+127
WHILE
        READ CC FROM F1
        AA=ER==C0


ON AA
        CHOOSE ON CC
        CASE OE;ACCEPT THE NEXT CHARACTER UNCRITICALLY
                READ CC FROM F1
                GOTO 77
        CASE RB;BEGIN A DEFFINITION
                ILP(INM)=III
                INM++
                ML=+000

        CASE NL;IGNORE
        CASE RC;IGNORE FOLLOWING COMMENTS AND MARK THE LINE END
                LS(III)=RC
                III++
                AA=ML<!MM
                IF AA;THIS LINE SHORTEST YET
                        MM=ML


                ENDIF
                WHILE
                        READ CC FROM F1
                        AA=CC!=NL


                ON AA
                ENDWHILE
        CASE OG;IF USING IGNORE, IGNORE
                IF UG
                ELSE
                        GOTO 77


                ENDIF
        DEFAULT;
        LOC 77
```

```
IMP++
AA=LS(IMP)
O2=AA
GOSUB CD;FOR DIGIT CONVERSION
DG=AA
IMP++
O3=LS(IMP); DESTINATION INDICATOR
IF UT
        WRITE O1
        WRITE O2
        WRITE O3
ENDIF
CHOOSE ON O1
CASE OP;FETCH PARAMETER
        ITU=IPR(DG);
CASE OD;CONVERT FROM CHAR TO DIGIT
        IAA=IPR(DG);
        AA=IAA
        GOSUB CD;DIGIT CONVERSION
        ITU=AA
CASE OB;POP STACK
        ITU=ISS(SP)
        AA=SP<=C0
        IF AA
                MS 'S STACKER'
                WRITE NL
                SP=C1
        ENDIF
        SP--;
CASE OS;FETCH FROM STACK WITHOUT POP
        ITU=ISS(SP)
CASE OH;FETCH AND WRITE HEX CONSTANT BYTE
        AA=O2
        GOSUB CH
        IAA=AA
        IAA=IAA*I16
        AA=O3
        GOSUB CH
        IBB=AA
        ITU=IAA+IBB
        O3=OC

CASE OL;LITERAL BYTE FETCH
        ITU=O2
CASE ON;LITERAL DIGIT FETCH
        AA=O2
        GOSUB CD
        ITU=AA
CASE OT;TURN ON TRACE MODE
        UT=+001
DEFAULT;FETCH A UNIQUE NUMBER
        ITU=IUU
        IUU++

ENDCHOOSE
IF UT
        III=ITU
        GOSUB PN
```

```
                LS(III)=CC
                III++
                AA=ILM<!III
                IF AA

                        MS 'MACMEMXST'
                        WRITE NL
                        CLOSE F1
                        STOP 5


                ENDIF
                ML++
        ENDCHOOSE

ENDWHILE
AA=CC!=EF
IF AA
        MS 'DEFN READ'
        STOP 2
ENDIF
CLOSE F1
IBC=IAV(C2)
TR='R'
OPEN F1 FOR TR AT IBC
IED=III;END OF DEFINITIONS
MS 'LOADED...'
ITU=III
GOSUB PN
MS '.BYTES FO'
MS 'R DEFINES'
WRITE NL
ILP(INM)=III
ITU=INM
GOSUB PN
MS '.MACROS..'
ITU=MM
GOSUB PN
MS ' MIN LEN.'
WRITE NL
ENDSUB;RM
                        SUB DM;DO MACRO EXPANSION


IMP=IJJ+I01
INM++
IDP=ILP(INM)
WHILE

        AA=IMP<!IDP
ON AA;  UNTIL WE HAVE READ UP TO THE NEXT MACRO DEFINITION
        AA=LS(IMP)
        IF UO
                AA=AA==MF
        ELSE
                AA=C0
        ENDIF
        IF AA;OPERATION CODE
                IMP++
                O1=LS(IMP);FROM  INDICATOR
```

(Continued on top of page 32)

```
                ITU=ISS(SP)
                GOSUB PN
                ITU=SP
                GOSUB PN
                ITU=III
                WRITE NL
        ENDIF
        CHOOSE ON O3
        CASE OC;CHARACTER OUTPUT
                AA=ITU
                WRITE AA INTO F2
        CASE OS;PUT ON THE STACK
                SP++
                AA=C4<=SP
                IF AA
                        MS 'S OVERFLW'
                        WRITE NL
                        SP=C4
                ENDIF
                ISS(SP)=ITU
        CASE OP;PUT INTO PARAMETER LOC
                IPR(DG)=ITU
        CASE OA;ADD TO STACK
                IAA=ISS(SP)
                IAA=IAA+ITU
                ISS(SP)=IAA
        CASE OR;SUBTRACT FROM STACK
                IAA=ISS(SP)
                IAA=IAA-ITU
                ISS(SP)=IAA
        CASE OM; MULTIPLY BY BASE AND ADD
                IAA=ISS(SP)
                IAA=IAA*I10
                IAA=IAA+ITU
                ISS(SP)=IAA
        CASE OH; OUTPUT HIGH BYTE
                UNPACK(ITU,AA,BB)
                WRITE AA INTO F2
        DEFAULT;WRITE OUT AS A DECIMAL NUMBER
                GOSUB WN
        ENDCHOOSE
    ELSE;END OF ACTION SECTION
    AA=LS(IMP)
    IF UN
            BB=AA!=RC
    ELSE
            BB=C1
    ENDIF
    IF BB
            WRITE AA INTO F2
    ELSE
            WRITE NL INTO F2
    ENDIF
    ENDIF
    IMP++
ENDWHILE
UT=+000;TURN OFF TRACE AT END OF MACRO
ENDSUB;DM
BOTTOM
```

end